Introduction to MPI

Kadin Tseng

Scientific Computing and Visualization Group

Boston University

Useful SCV Info

- SCV home page http://scv.bu.edu/
- Resource Applications
  - http://scv.bu.edu/accounts/
- Help
  - Online FAQs (http://scv.bu.edu/)
  - Web-based tutorials (MPI, OpenMP, MATLAB, Graphics tools)
  - HPC consultations by appointment
    - **Kadin Tseng (kadin@bu.edu), Doug Sondak (sondak@bu.edu)**
  - help@twister.bu.edu, help@cootie.bu.edu

Parallel Computing Paradigms

- Parallel Computing Paradigms
  - Message Passing (MPI, PVM, ...)
    - Distributed or shared memory
  - Directives (OpenMP, ...)
    - Shared memory
  - Multi-Level Parallel programming (MPI + OpenMP)
    - Shared (and distributed) memory

MPI Topics to Cover

- Fundamentals
- Basic MPI Functions
- Point-to-point Communications
- Compilation and Execution for pSeries, P3, Blue Gene
- Collective Communications
- Dynamic Memory Allocations
- MPI Timer
- Cartesian Topology
- User-defined Reduction Operations

What is MPI ?

- MPI stands for Message Passing Interface.
- It is a library of subroutines/functions, not a computer language.
- These subroutines/functions are callable from fortran (F77, F9x) or C (C, C++) programs.
- Programmer writes fortran/C code, insert appropriate MPI subroutine/function calls, compile and finally link with MPI message passing library.
- In general, MPI codes run on shared-memory multi-processors, distributed-memory multi-computers, cluster of workstations, or heterogeneous clusters of the above.
- At BU, many MPI-2 functionalities are available on the IBM pSeries. Not so on the Bluegene or Linux Cluster.

Why MPI ?

- To provide efficient communication (message passing) among networks/clusters of nodes
- To enable more analyses in a prescribed amount of time.
- To reduce time required for one analysis.
- To increase fidelity of physical modeling.
- To have access to more memory.
- To enhance code portability; works for both shared- and distributed-memory.
- For “embarrassingly parallel” problems, such as many Monte-Carlo applications, parallelizing with MPI can be trivial with near-linear (or even superlinear) speedup.

MPI Preliminaries

- MPI's pre-defined constants, function prototypes, etc., are included in a header file. This file must be included in your code wherever MPI function calls appear (in "main" and in user subroutines/functions) :
  - #include "mpi.h" for C codes
  - #include "mpi++.h" * for C++ codes
  - include "mpif.h" for f77 and f9x codes
- MPI_Init must be the first MPI function called.
- Terminates MPI by calling MPI_Finalize.
- These two functions must only be called once in user code.

* More on this later …

MPI Preliminaries (continued)

- C is case-sensitive language. MPI function names always begin with “MPI_”, followed by specific name with leading character capitalized, *e.g.*, MPI_Comm_rank. MPI pre-defined constant variables are expressed in upper case characters, *e.g.,* MPI_COMM_WORLD.
- Fortran is not case-sensitive. No specific case rules apply.
- MPI fortran routines return error status as last argument of subroutine call, *e.g.,*

  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
- Error status is returned as “int” function value for C MPI functions, *e.g.,*

  int ierr = MPI_Comm_rank(MPI_COMM_WORLD, rank);

What is A Message ?

- Collection of data (array) of MPI data types
  - Basic data types such as int /integer, float/real
  - Derived data types
- Message “envelope” – source, destination, tag, communicator

Modes of Communication

- Point-to-point communication
  - Blocking – returns from call when task completes
    - Several send modes; one receive mode
  - Nonblocking – returns from call without waiting for task to complete
    - Several send modes; one receive mode
- Collective communication

Essentials of Communication

- Sender must specify valid destination.
- Sender and receiver data type, tag, communicator must match.
- Receiver can receive from non-specific (but valid) source.
- Receiver returns extra (status) parameter to report info regarding message received.
- Sender specifies size of sendbuf; receiver specifies *upper bound* of recvbuf.

MPI Data Types vs C Data Types

- MPI types -- C types
  - MPI_INT – signed int
  - MPI_UNSIGNED – unsigned int
  - MPI_FLOAT – float
  - MPI_DOUBLE – double
  - MPI_CHAR – char
  - . . .

MPI vs Fortran Data Types

- MPI_INTEGER – INTEGER
- MPI_REAL – REAL
- MPI_DOUBLE_PRECISION – DOUBLE PRECISION
- MPI_CHARACTER – CHARACTER(1)
- MPI_COMPLEX – COMPLEX
- MPI_LOGICAL – LOGICAL
- . . .

MPI Data Types

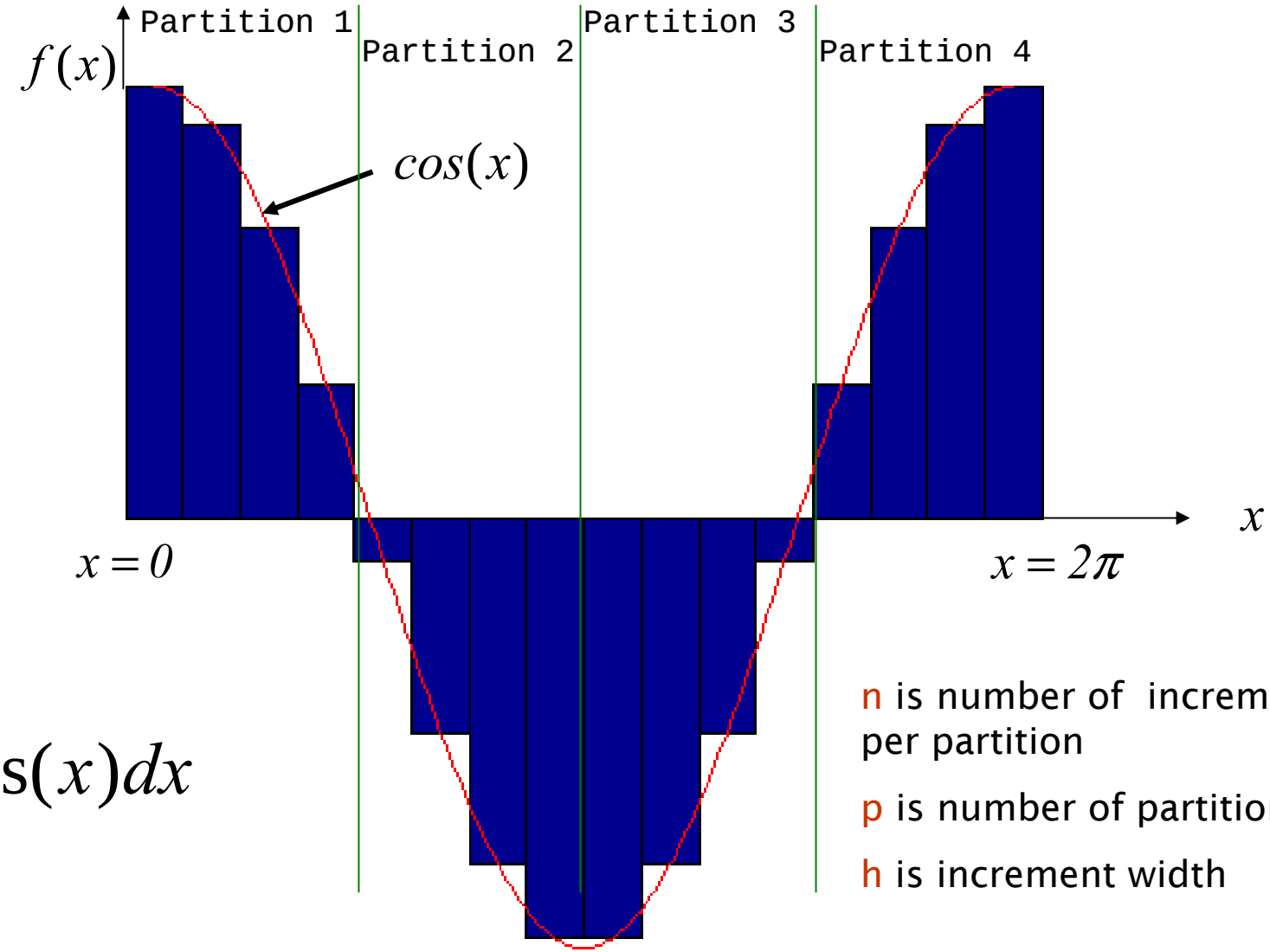
- MPI_PACKED
- MPI_BYTE
- User-derived types

Some MPI Implementations

There are a number of implementations :

- MPICH (ANL)
  - Version 1.2 is latest.
  - There is a list of supported MPI-2 functionalities.
- LAM (UND/OSC)
- CHIMP (EPCC)
- Vendor implementations (SGI, IBM, …)
- Don't worry! - as long as vendor supports the MPI Standard (IBM's MPL does not).
- Job execution procedures of implementations may differ.
- MPICH2 (MPI-2) is now available from ANL.

Integrate cos(x) by Mid-point Rule



$$\int_0^{2\pi} \cos(x)\,dx$$

n is number of increments per partition

p is number of partitions

h is increment width

Example 1 (Integration)

We will introduce some fundamental MPI function calls through the computation of a simple integral by the Mid-point rule.

$$\int_a^b \cos(x)dx = \sum_{i=0}^{p-1}\sum_{j=0}^{n-1}\int_{a_{ij}}^{a_{ij}+h} \cos(x)dx$$

$$\approx \sum_{i=0}^{p-1}\left[\sum_{j=0}^{n-1}\cos(a_{ij}+\tfrac{h}{2})h\right]$$

$$h=(b-a)/\,p/\,n; \qquad a_{ij}=a+(i*n+j)*h$$

p is number of partitions and n is increments per partition

Example 1 - Serial fortran code


```
Program Example1
implicit none
integer n, p, i, j
real h, integral_sum, a, b, integral, pi
pi = acos(-1.0)  !  = 3.14159...
a = 0.0             ! lower limit of integration
b = pi/2.           ! upper limit of integration
p = 4               ! number of partitions (processes)
n = 500             ! number of increments in each partition
h = (b-a)/p/n    ! length of increment
integral_sum = 0.0      ! Initialize solution to the integral
do i=0,p-1          ! Integral sum over all partitions
   integral_sum = integral_sum + integral(a,i,h,n)
enddo
print *,'The Integral =', integral_sum
stop
end
```

. . Serial fortran code (cont'd)

example1.f continues . . .

```
      real function integral(a, i, h, n)
! This function computes the integral of the ith partition
      implicit none
      integer n, i, j     ! i is partition index; j is increment index
      real h, h2, aij, a

      integral = 0.0    ! initialize integral
      h2 = h/2.
      do j=0,n-1                  ! sum over all "j" integrals
        aij = a + (i*n + j)*h     ! lower limit of integration of "j"
        integral = integral + cos(aij+h2)*h    ! contribution due "j"
      enddo

      return
      end
```

Example 1 - Serial C code


```
#include <math.h>
#include <stdio.h>
float integral(float a, int i, float h, int n);
void main() {
    int n, p, i, j, ierr;
    float h, integral_sum, a, b, pi;
    pi = acos(-1.0);  /* = 3.14159... *
    a = 0.;              /* lower limit of integration */
    b = pi/2.;           /* upper limit of integration */
    p = 4;               /* # of partitions */
    n = 500;             /* increments in each process */
    h = (b-a)/n/p;    /* length of increment */
    integral_sum = 0.0;
    for (i=0; i<p; i++) { /* integral sum over partitions */
      integral_sum += integral(a,i,h,n);
    }
    printf("The Integral =%f\n", integral_sum);
}
```

. . Serial C code (cont'd)

example1.c continues . . .

```
float integral(float a, int i, float h, int n) {
  int j;
  float h2, aij, integ;
  integ = 0.0;                  /* initialize integral */
  h2 = h/2.;
  for (j=0; j<n; j++) {         /* sum over integrals in partition i*/
    aij = a + (i*n + j)*h;      /* lower limit of integration of j*/
    integ += cos(aij+h2)*h;  /* contribution due j */
  }
  return integ;
}
```

Example 1_1 - Parallel f77 code

Two main styles of programming: SPMD, MPMD. The following demonstrates SPMD, which is more frequently used than MPMD,

MPI functions used in this example:

- MPI_Init, MPI_Comm_rank, MPI_Comm_size
- MPI_Send, MPI_Recv, MPI_Finalize

```
PROGRAM Example1_1
 implicit none
 integer n, p, i, j, ierr, master, myid
 real h, integral_sum, a, b, integral, pi
 include "mpif.h"   ! pre-defined MPI constants, ...
 integer source, tag, status(MPI_STATUS_SIZE)
 real my_int

 data master/0/   ! 0 is the master processor  responsible
                  ! for collecting integral sums …
```

. . . Parallel fortran code (cont'd)

```
! Starts MPI processes ...
    call MPI_Init(ierr)
! Get current process id
    call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
! Get number of processes from command line
    call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)

! executable statements before MPI_Init is not
! advisable; side effect implementation-dependent   (historical)

   pi = acos(-1.0)     ! = 3.14159...
   a = 0.0             ! lower limit of integration
   b = pi/2.           ! upper limit of integration
   n = 500             ! number of increments in each process
   h = (b - a)/ p / n  ! (uniform) increment size
   tag = 123           ! set tag for job
```

... Parallel fortran code (cont'd)


```
  my_int = integral(a, myid, h, n)    ! compute local sum due myid
  write(*,"('Process ',i2,' has the partial integral of',
&       f10.6)")myid,my_int
  call MPI_Send(my_int, 1, MPI_REAL, master, tag,
&     MPI_COMM_WORLD, ierr)    ! send my_int to master

  if(myid .eq. master) then
     do source=0,p-1    ! loop on all procs to collect local sum (serial !)
        call MPI_Recv(my_int, 1, MPI_REAL, source, tag,
&             MPI_COMM_WORLD, status, ierr)   ! not safe
        integral_sum = integral_sum + my_int
     enddo
     print *,'The Integral =', integral_sum
  endif
  call MPI_Finalize(ierr)                ! let MPI finish up
end
```

Message Passing to Self

- It is valid to send/recv message to/from itself
- On IBM pSeries, env variable MP_EAGER_LIMIT may be used to control buffer memory size.
- Above example hangs if MP_EAGER_LIMIT set to 0
- Good trick to use to see if code is “safe”
- Not available with MPICH

Example 1_2 - Parallel C code


```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
float  integral(float a, int i, float h, int n); /* prototype */
void main(int argc, char *argv[]) {
   int n, p, i;
   float h, result, a, b, pi, my_int;
   int myid, source, master, tag;
   MPI_Status status;                  /* MPI data type */
   MPI_Init(&argc, &argv);             /* start MPI processes */
   MPI_Comm_rank(MPI_COMM_WORLD, &myid); /* current proc. id */
   MPI_Comm_size(MPI_COMM_WORLD, &p);      /* # of processes */
```

… Parallel C code (continued)


```
  pi = acos(-1.0);     /* = 3.14159... */
  a = 0.;              /* lower limit of integration */
  b = pi/2.;           /* upper limit of integration */
  n = 500;             /* number of increment within each process */
  master = 0;
/* define the process that computes the final result */
  tag = 123;           /* set the tag to identify this particular job */
  h = (b-a)/n/p;       /* length of increment */
  my_int = integral(a,myid,h,n);   /* local sum due process myid */
  printf("Process %d has the partial integral of %f\n", myid,my_int);
```

… Parallel C code (continued)


```
if(myid == 0) {
   integral_sum = my_int;
   for (source=1;source<p;i++) {
       MPI_Recv(&my_int, 1, MPI_FLOAT, source, tag,
           MPI_COMM_WORLD, &status);  /* safe */
     integral_sum += my_int;
   }
   printf("The Integral =%f\n", integral_sum);
 } else {
   MPI_Send(&my_int, 1, MPI_FLOAT, master, tag,
         MPI_COMM_WORLD);  /* send my_int to "master" */
 }
   MPI_Finalize();      /* let MPI finish up … */
}
```

Compilation & Execution

In the following slides, the compilation and job running procedures will be outlined for the three computer systems maintained by SCV:

- IBM pSeries 655 and 690
- IBM Bluegene/L
- Linux Cluster

How To Compile On pSeries

On AIX:

- `Twister % mpxlf example.f` (F77)
- `Twister % mpxlf90 example.f` (F90)
- `Twister % mpxlf90 example.f90` (F90)
- `Twister % mpcc example.c` (C)
- `Twister % mpCC -D_MPI_CPP_BINDINGS example.C` (C++)

See
http://scv.bu.edu/computation/pseries/programming.html

The above compiler scripts should be used for MPI code compilation as they automatically include appropriate include files (–I) and library files (–L) for successful compilations.

How To Run Jobs On pSeries

Interactive jobs:

- `Twister % a.out –procs 4  or`
- `Twister % poe a.out –procs 4`

LSF batch jobs:

- `Twister % bsub –q queue-name “a.out –procs 4”`

See
http://scv.bu.edu/computation/pseries/runningjobs.html

Output of Example1_1


```
skate% mpicc –o example1_1 example1_1.c -lm
Skate% qsub –I –l nodes=2:ppn=2
. . .
node012% mpijob example1_1
Process 1 has the partial result of  0.324423
Process 2 has the partial result of  0.216773
Process 0 has the partial result of  0.382683
Process 3 has the partial result of  0.076120
  The Integral =   1.000000
```

Processing out of order !

How To Compile On Bluegene

BGL consists of front-end and back-end. Compilation is performed on the FE but job is run on the BE. A cross compiler is required to achieve this:

- `Lee % blrts_xlf example.f …` (F77)
- `Lee % blrts_xlf90 example.f …` (F90)
- `Lee % blrts_xlf90 example.f90 …` (F90)
- `Lee % blrts_xlc example.c …` (C)
- `Lee % blrts_xlC –D_MPI_CPP_BINDINGS example.C …` (C++)

Need to link-in a handful of libraries, include files, etc., compilation is best handled with a makefile. For details, consult
http://scv.bu.edu/computation/bluegene/programming.html
Many of the compiler switches are the same as for AIX. However, DO NOT use the –qarch=auto.

How To Run Jobs On Bluegene

Interactive job: Not permitted

Loadleveler batch:

- Lee % llsubmit user-batch-script or
- Lee % bglsub nprocs CWD EXE [“more MPI args”]

```
(A user script file called bglsub.$USER will also be generated. You can also use that along with llsubmit to run job)
Example:
Lee % bglsub 32 $PWD $PWD/example1_4 “< mystdin”
```

For details, see

http://scv.bu.edu/computation/bluegene/runningjobs.html

How To Compile On Linux (1)

On Linux Cluster:

- Skate % mpif77 example.f (F77)
- Skate % mpif90 example.f (F90)
- Skate % mpicc example.c (C)
- Skate % mpiCC example.C (C++)

See http://scv.bu.edu/computation/linuxcluster/programming.html

- The above scripts should be used for MPI code compilation as they automatically include appropriate include files (–I) and library files (–L) for successful compilations.
- Above script names are generic. Currently, compilers available include: Gnu and Intel. Multiple versions of the same compiler may exist. Use the **module** command to query or reset selections.

How To Compile On Linux (2)


```
cootie% module list
Currently Loaded Modulefiles:
1)  gcc/3.4.5(default)
2)  intel/9.0(default)
3)  mpichgm/1.2.6..14b_gcc(default:mpich-gcc)

Other module options include: load, unload, avail

cootie% module unload mpich
cootie% module load mpich-intel
cootie% module list
1)  gcc/3.4.5(default)
2)  intel/9.0(default)
3)  mpichgm/1.2.6..14b_intel(mpich-intel)
```

How To Compile On Linux (3)

- Alternatively, set up makefile for compilation
- Use module as described
- Or provide absolute path to the preferred compiler script as in below

```
ROOT=/usr/local/IT/mpichgm-1.2.6..14b/intel/bin/
OPTFLAGS   = -O3
CC         = $(ROOT)mpicc $(OPTFLAGS)
CCC        = $(ROOT)mpiCC $(OPTFLAGS)
F77        = $(ROOT)mpif77 $(OPTFLAGS)
F90        = $(ROOT)mpif90 $(OPTFLAGS)
```

Header file for C++

On IBM p655 and p690: use mpi.h and –DHAVE_MPI_CXX
On Bluegene/L: use mpi.h and –D_MPI_CPP_BINDINGS
On the Linux Cluster, use mpi++.h

If you want to use the same source code on all three systems, you could use CPP to automatically invoke appropriate C++ header file:

```
#ifdef _MPI_CPP_BINDINGS
    #include <mpi.h>
#elif HAVE_MPI_CXX
   #include <mpi.h>
#else
   #include <mpi++.h>
#endif
#include <iostream.h>
#include <math.h>
#include <stdio.h>
int main(int argc, char* argv[])
{
   . . . . . .
```

Run Jobs on Linux Cluster

- Please use batch to run single-processor jobs whenever possible to avoid burdens on interactive nodes `skate` and `cootie`
- Unlike AIX, Cluster batch system is PBS (Portable Batch System)
  - `man pbs` for detail
  - Most often used pbs commands :
    - `qsub` – job submission (like `bsub` for LSF)
    - `qstat` – query for jobs status (like `bjobs` for LSF)
    - `qdel` – kill job (like `bkill` for LSF)

```
% qsub my_pbs_script
```

`my_pbs_script` is a user-script containing user-specific pbs batch job info.

my_pbs_script (1)


```
#!/bin/bash
# Set the default queue
#PBS -q dque
# Request 4 nodes with 1 processor per node
#PBS -l nodes=4:ppn=1
# Set wallclock limit to 10 minutes
#PBS –l walltime=00:10:00
# prefer to use the Intel-compiled mpich
module unload mpich
module load mpich-intel
MYPROG="psor"
GMCONF=`/usr/local/pbs/util/mknodes`
/usr/local/xcat/bin/pbsnodefile2gmconf $PBS_NODEFILE >$GMCONF
cd $PBS_O_WORKDIR
NP=`wc -l $GMCONF |awk '{print $1}'`
mpirun -machinefile $GMCONF -np $NP PBS_JOBID=$PBS_JOBID $MYPROG
exit 0
```

my_pbs_script (2)

Alternatively …

```
. . .
#PBS –l walltime=00:10:00
module unload mpich
module load mpich-intel
MYPROG="psor"
GMCONF=`/usr/local/pbs/util/mknodes`
. . .
NP=`wc -l $GMCONF |awk '{print $1}'`
/usr/local/IT/mpichgm-1.2.6..14b/intel/bin/mpirun
-machinefile $GMCONF -np $NP PBS_JOBID=$PBS_JOBID $MYPROG
exit 0
```

Interactive batch (1)

What is single processor interactive batch for ?

Jobs that requires more than a few minutes to run at the monitor.

- Compute-intensive Mathematica, MATLAB jobs.
- Debugging codes.

How to initiate an interactive PBS batch job

```
cootie % qsub –I
```

If nodes are available, one will be assigned to your request and you will be logged into that node – at your home directory. Proceed to appropriate directory to perform tasks. When done, be sure to exit to terminate interactive batch job.

Interactive batch (2)

What is multiprocessor interactive batch jobs for ?

Debugging MPI jobs.

How to initiate an interactive multiprocessor PBS batch job ?

```
cootie % qsub –I –l nodes=2:ppn=2
```

Above batch requests 4 processors. If requested number of nodes are available, they will be assigned to you and you will be logged into one in the group of nodes – at your home directory. Proceed to appropriate directory to perform parallel task. When done, be sure to exit to terminate interactive batch job and return to original window.

For an MPI job, need to know which nodes are assigned -- not known a priori. I wrote a handy script called mpijob:

```
node034% mpijob a.out [-np nprocs]
```

See http://scv.bu.edu/computation/linuxcluster/runningjobs.html for further details.

Example1_3 – Parallel Integration

MPI functions used for this example:

- MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Finalize
- MPI_Recv, MPI_Isend, MPI_Wait
- MPI_ANY_SOURCE, MPI_ANY_TAG

```
PROGRAM Example1_3
    implicit none
    integer n, p, i, j, k, ierr, master
    real h, a, b, integral, pi
    integer req(1)
    include "mpif.h"    ! This brings in pre-defined MPI constants, ...
    integer myid, source, dest, tag, status(MPI_STATUS_SIZE)
    real my_int, integral_sum

    data master/0/
```

Example1_3 (continued)


```
c**Starts MPI processes ...
      call MPI_Init(ierr)
      call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
      call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)

      pi = acos(-1.0)     !  = 3.14159...
      a = 0.0             ! lower limit of integration
      b = pi/2.           ! upper limit of integration
      n = 500             ! number of increment within each process
      dest = master       ! define process that computes the final result
      tag = 123           ! set the tag to identify this particular job
      h = (b-a)/n/p       ! length of increment

      my_int = integral(a,myid,h,n)     ! Integral of process myid
      write(*,*)'myid=',myid,',  my_int=',my_int
```

Example1_3 (continued)


```
      if(myid .eq. master) then                    ! the following serialized
        integral_sum = my_int
        do k=1,p-1
          call MPI_Recv(my_int, 1, MPI_REAL,
     &       MPI_ANY_SOURCE, MPI_ANY_TAG,  ! more efficient and
     &       MPI_COMM_WORLD, status, ierr)      ! less prone to deadlock
          integral_sum = integral_sum + my_int      ! sum of local integrals
        enddo
      else
        call MPI_Isend(my_int, 1, MPI_REAL, dest, tag,
     &     MPI_COMM_WORLD, req, ierr)   ! send my_int to "dest"
C**more computation here . . .
        call MPI_Wait(req, status, ierr)       ! wait for nonblock send ...
      endif
c**results from all procs have been collected and summed ...
      if(myid .eq. 0) write(*,*)'The Integral =',integral_sum
      call MPI_Finalize(ierr)                        ! let MPI finish up ...
      stop
      end
```

Example1_4 Parallel Integration

MPI functions and constants used for this example:

- MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Finalize
- MPI_Bcast, MPI_Reduce, MPI_SUM

```
PROGRAM Example1_4
implicit none
integer n, p, i, j, ierr, master
real h, integral_sum, a, b, integral, pi

include "mpif.h"     ! This brings in pre-defined MPI constants, ...
integer myid, source, dest, tag, status(MPI_STATUS_SIZE)
real my_int

data master/0/
```

Example1_4 (continued)


```
c**Starts MPI processes ...
      call MPI_Init(ierr)
      call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
      call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)

      pi = acos(-1.0)   !  = 3.14159...
      a = 0.0           ! lower limit of integration
      b = pi/2.         ! upper limit of integration
      h = (b-a)/n/p     ! length of increment
      dest = 0          ! define the process that computes the final result
      tag = 123         ! set the tag to identify this particular job
      if(myid .eq. master) then
        print *,'The requested number of processors =',p
        print *,'enter number of increments within each process'
        read(*,*)n
      endif
```

Example1_4 (continued)


```
c**Broadcast "n" to all processes
      call MPI_Bcast(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
      my_int = integral(a,myid,h,n)
      write(*,"('Process ',i2,' has the partial sum of',f10.6)")
     &         myid, my_int

      call MPI_Reduce(my_int, integral_sum, 1, MPI_REAL, MPI_SUM,
     &           dest, MPI_COMM_WORLD, ierr)  ! Compute integral sum

      if(myid .eq. master) then
        print *,'The Integral Sum =', integral_sum
      endif

      call MPI_Finalize(ierr)                    ! let MPI finish up ...
      stop
      end
```

Example1_5 Parallel Integration

New MPI functions and constants used for this example:

- MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Finalize
- MPI_Pack, MPI_Unpack
- MPI_FLOAT_INT, MPI_MINLOC, MPI_MAXLOC, MPI_PACKED

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
float fct(float x) { return cos(x); }
/* Prototype */
float integral(float a, int i, float h, int n);
int main(int argc, char* argv[])
{
```

Example1_5 (cont'd)


```
int n, p;
float h,integral_sum, a, b, pi;
int myid, dest, m, index, minid, maxid, Nbytes=1000, master=0;
char line[10], scratch[Nbytes];
struct {
        float val;
        int   loc; } local_sum, min_sum, max_sum;

MPI_Init(&argc,&argv);                              /* starts MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* process id */
MPI_Comm_size(MPI_COMM_WORLD, &p);       /* num of procs*/
pi = acos(-1.0);  /* = 3.14159... */
dest = 0;         /* define the process to compute final result */
comm = MPI_COMM_WORLD;
```

Example1_5 (cont'd)


```
if(myid == master) {
        printf("The requested number of processors = %d\n",p);
        printf("enter number of increments within each process\n");
        (void) fgets(line, sizeof(line), stdin);
        (void) sscanf(line, "%d", &n);
        printf("enter a & m\n");
        printf(" a = lower limit of integration\n");
        printf(" b = upper limit of integration\n");
        printf("   = m * pi/2\n");
        (void) fgets(line, sizeof(line), stdin);
        (void) sscanf(line, "%d %d", &a, &m);
        b = m * pi / 2.;
}
```

Example1_5 (cont'd)


```
 If (myid == master) {
/* to be efficient, pack all things into a buffer for broadcast */
      index = 0;
      MPI_Pack(&n, 1, MPI_INT,   scratch, Nbytes, &index, comm);
      MPI_Pack(&a, 1, MPI_FLOAT, scratch, Nbytes, &index, comm);
      MPI_Pack(&b, 1, MPI_FLOAT, scratch, Nbytes, &index, comm);
      MPI_Bcast(scratch, Nbytes, MPI_PACKED, master, comm);
  } else {
      MPI_Bcast(scratch, Nbytes, MPI_PACKED, master, comm);
/* things received have been packed, unpack into expected locations */
      index = 0;
      MPI_Unpack(scratch, Nbytes, &index, &n, 1, MPI_INT,   comm);
      MPI_Unpack(scratch, Nbytes, &index, &a, 1, MPI_FLOAT, comm);
      MPI_Unpack(scratch, Nbytes, &index, &b, 1, MPI_FLOAT, comm);
    }
```

Example1_5 (cont'd)


```
    h = (b-a)/n/p;     /* length of increment */
    local_sum.val = integral(a,myid,h,n);
    local_sum.loc = myid;

printf("Process %d has the partial sum of %f\n", myid, local_sum.val);

/* data reduction with MPI_SUM */
    MPI_Reduce(&local_sum.val, &integral_sum, 1, MPI_FLOAT,
MPI_SUM, dest, comm);

/* data reduction with MPI_MINLOC */
    MPI_Reduce(&local_sum, &min_sum, 1, MPI_FLOAT_INT,
MPI_MINLOC, dest, comm);

/* data reduction with MPI_MAXLOC */
    MPI_Reduce(&local_sum, &max_sum, 1, MPI_FLOAT_INT,
MPI_MAXLOC, dest, comm);
```

Example1_5 (cont'd)

```
 if(myid == master) {
       printf("The Integral = %f\n", integral_sum);
       maxid = max_sum.loc;
       printf("Proc %d has largest integrated value of %f\n",maxid,
max_sum.val);
       minid = min_sum.loc;
       printf("Proc %d has smallest integrated value of %f\n", minid,
min_sum.val);
     }

     MPI_Finalize();                    /* let MPI finish up ... */
}
```

C++ example

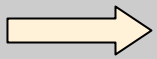

```
#include <mpi.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
  int rank, size;
  MPI::Init(argc, argv);
  rank = MPI::COMM_WORLD.Get_rank();
  size = MPI::COMM_WORLD.Get_size();
  cout << "Hello world! I am " << rank <<
          "of " << size << endl;
  MPI::Finalize();
  return 0; }

Twister % mpCC –DHAVE_MPI_CXX –o hello hello.C
Twister % hello –procs 4
```

Collective Communications

Sometimes, one wishes to gather data from one or more processes and share among all participating processes. At other times, one may wish to distribute data from one or more processes to a specific group of processes. Previously we have introduced an MPI routine `MPI_Bcast` to broadcast data from one process to all other participating processes. Broadcasting and the gather/scatter communications alluded to above are called collective communications. These routines and their functionalities are shown diagramatically in the table below. The first four columns on the left denote the contents of respective send buffers (*e.g.,* arrays) of four processes. The content of each buffer, shown here as alphabets, is assigned a unique color to identify its origin. For instance, the alphabets in blue indicate that they originate from process 1. The middle column shows the MPI routines with which the send buffers are operated on. The four columns on the right represent the contents of the processes' receive buffers resulting from the MPI operations.

Collective Functions

Process 0	Process 1*	Process 2	Process 3	Operation	Process 0	Process 1*	Process 2	Process 3
	b			MPI_Bcast	b	b	b	b
a	b	c	d	MPI_Gather		a,b,c,d		
a	b	c	d	MPI_Allgather	a,b,c,d	a,b,c,d	a,b,c,d	a,b,c,d
	a,b,c,d			MPI_Scatter	a	b	c	d
a,b,c,d	e,f,g,h	i,j,k,l	m,n,o,p	MPI_Alltoall	a,e,i,m	b,f,j,n	c,g,k,o	d,h,l,p
SendBuff	SendBuff	SendBuff	SendBuff		ReceiveBuff	ReceiveBuff	ReceiveBuff	ReceiveBuff

- *This example uses 4 processes*
- *Rank 1 is, arbitrarily, designated data gather/scatter process*
- *a, b, c, d are scalars or arrays of any data type*
- *Data are gathered/scattered according to rank order*

Collectives Example Code

```
      program collectives_example
      implicit none
      integer p, ierr, i, myid, root
      include "mpif.h"        ! This brings in pre-defined MPI constants, ...
      character*1 x(0:3), y(0:3), alphabets(0:15)
      data alphabets/'a','b','c','d','e','f','g','h','i','j','k','l',
     &               'm','n','o','p'/
      data root/1/            ! process 1 is the data sender/receiver
c**Starts MPI processes ...
      call MPI_Init(ierr)     ! starts MPI
      call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr) ! current pid
      call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)    ! # of procs
```

Collectives Example (cont'd)


```
 if (myid .eq. 0) then
    write(*,*)
    write(*,*)'* This program demonstrates the use of collective',
&               ' MPI functions'
    write(*,*)'* Four processors are to be used for the demo'
    write(*,*)'* Process 1 (of 0,1,2,3) is the designated root'
    write(*,*)
    write(*,*)
    write(*,*)' Function Proc Sendbuf Recvbuf'
    write(*,*)' -------- ---- ------- -------'
 endif
```

Gather Operation


```
alphabets(0) = 'a'
alphabets(1) = 'b'

  . . .

alphabets(14) = 'o'
alphabets(15) = 'p'
```

```
c**Performs a gather operation
      x(0) = alphabets(myid)
      do i=0,p-1
         y(i) = ' '
      enddo
      call MPI_Gather(x,1,MPI_CHARACTER,  ! Send-buf,count,type,
     &                y,1,MPI_CHARACTER,  ! Recv-buf,count?,type?,
     &                root,               ! Data destination
     &                MPI_COMM_WORLD,ierr)    ! Comm, flag
      write(*,"('MPI_Gather:',t20,i2,(3x,a1),t40,4(3x,a1))")myid,x(0),y
```

Recv-buf according to rank order

All-gather Operation


```
c**Performs an all-gather operation
      x(0) = alphabets(myid)
      do i=0,p-1
         y(i) = ' '
      enddo
      call MPI_Allgather(x,1,MPI_CHARACTER,      ! send buf,count,type
     &                   y,1,MPI_CHARACTER,      ! recv buf,count,type
     &                   MPI_COMM_WORLD,ierr)    ! comm,flag
      write(*,"('MPI_Allgather:',t20,i2,(3x,a1),t40,4(3x,a1))")myid,x(0),y
```

Scatter Operation


```
c**Perform a scatter operation
      if (myid .eq. root) then
        do i=0, p-1
           x(i) = alphabets(i)
           y(i) = ' '
        enddo
      else
         do i=0,p-1
            x(i) = ' '
            y(i) = ' '
         enddo
      endif
      call MPI_scatter(x,1,MPI_CHARACTER,        ! Send-buf,count,type
     &                 y,1,MPI_CHARACTER,        ! Recv-buf,count,type
     &                 root,                     ! data origin
     &                 MPI_COMM_WORLD,ierr)      ! comm,flag
      write(*,"('MPI_scatter:',t20,i2,4(3x,a1),t40,4(3x,a1))")myid,x,y
```

Alltoall Operation


```
c**Perform an all-to-all operation
      do i=0,p-1
         x(i) = alphabets(i+myid*p)
         y(i) = ' '
      enddo
      call MPI_Alltoall(x,1,MPI_CHARACTER,        ! send buf,count,type
     &                  y,1,MPI_CHARACTER,        ! recv buf,count,type
     &                  MPI_COMM_WORLD,ierr)      ! comm,flag
      write(*,"('MPI_Alltoall:',t20,i2,4(3x,a1),t40,4(3x,a1))")myid,x,y
```

Broadcast Operation


```
c**Performs a broadcast operation
      do i=0, p-1
         x(i) = ' '
         y(i) = ' '
      enddo
      if(myid .eq. root) then
         x(0) = 'b'
         y(0) = 'b'
      endif
      call MPI_Bcast(y,1,MPI_CHARACTER,              ! buf,count,type
     &               root,MPI_COMM_WORLD,ierr)  ! root,comm,flag
       write(*,"('MPI_Bcast:',t20,i2,4(3x,a1),t40,4(3x,a1))")myid,x,y

       call MPI_Finalize(ierr)     ! let MPI finish up ...
       end
```

Example 1.6 Integration (modified)

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
float fct(float x)
{
     return cos(x);
}
/* Prototype */
float integral(float a, int i, float h, int n);
int main(int argc, char* argv[])
{
int n, p, myid, i;
     float h, integral_sum, a, b, pi, my_int;
     float buf[50], tmp;
```

Example 1.6 (cont'd)


```
MPI_Init(&argc,&argv);                /* starts MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* current proc id */
MPI_Comm_size(MPI_COMM_WORLD, &p);     /* num of procs */

pi = acos(-1.0);  /* = 3.14159... */
a = 0.;           /* lower limit of integration */
b = pi*1./2.;     /* upper limit of integration */
n = 500;          /* number of increment within each process */
h = (b-a)/n/p;    /* length of increment */

my_int = integral(a,myid,h,n);

printf("Process %d has the partial sum of %f\n", myid,my_int);

MPI_Gather(&my_int, 1, MPI_FLOAT, buf, 1, MPI_FLOAT, 0,
MPI_COMM_WORLD);
```

Example 1.6 (cont'd)


```
 MPI_Scatter(buf, 1, MPI_FLOAT, &tmp, 1, MPI_FLOAT, 0,
MPI_COMM_WORLD);
    printf("Result sent back from buf = %f\n", tmp);

    if(myid == 0) {
      integral_sum = 0.0;
      for (i=0; i<p; i++) {
        integral_sum += buf[i];
      }
      printf("The Integral =%f\n", integral_sum);
    }

    MPI_Finalize();                    /* let MPI finish up ... */
}
```

MPI_Probe, MPI_Wtime (f90)

This example demonstrates dynamic memory allocation and parallel timer.

```
Program dma_example
implicit none
include "mpif.h"
integer, parameter :: real_kind = selected_real_kind(8,30)
real(real_kind), dimension(55) :: sdata
real(real_kind), dimension(:), allocatable :: rdata
real(real_kind) :: start_time, end_time
integer :: p, i, count, myid, n, status(MPI_STATUS_SIZE), ierr

!* Starts MPI processes ...
call MPI_Init(ierr)                                !* starts MPI
call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)  ! myid
call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)   ! Num. proc
```

MPI_Probe, MPI_Wtime (f90 cont'd)

```
start_time = MPI_Wtime()      ! start timer, measured in seconds
if (myid == 0) then
  sdata(1:50)= (/ (i, i=1,50) /)
  call MPI_Send(sdata, 50, MPI_DOUBLE_PRECISION, 1, 123, &
                MPI_COMM_WORLD, ierr)
else
  call MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD, status,
ierr)
  call MPI_Get_count(status, MPI_DOUBLE_PRECISION, count, ierr)
  allocate( rdata(count) )
  call MPI_Recv(rdata, count, MPI_DOUBLE_PRECISION, 0, &
                MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
  write(*,'(5f10.2)')rdata(1:count:10)
endif
end_time = MPI_Wtime()        ! stop timer
```

MPI_Probe, MPI_Wtime (f90 cont'd)

```
if (myid .eq. 1) then
  WRITE(*,"('    Total cpu time =',f10.5,' x ',i3)") end_time -
start_time,p
endif

call MPI_Finalize(ierr)                    !* let MPI finish up ...

end program dma_example
```

MPI_Probe, MPI_Wtime (C)

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
     double sdata[55], *rdata, start_time, end_time;
     int p, i, count, myid, n;
     MPI_Status status;

/* Starts MPI processes ... */
     MPI_Init(&argc, &argv);              /* starts MPI */
     MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get
current process id */
     MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number
of processes */
```

MPI_Probe, MPI_Wtime (C cont'd)


```
start_time = MPI_Wtime();   /* starts timer */
if (myid == 0) {
      for(i=0;i<50;++i) { sdata[i]=(double)i; }

MPI_Send(sdata,50,MPI_DOUBLE,1,123,MPI_COMM_WORLD);
} else {
      MPI_Probe(0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
      MPI_Get_count(&status,MPI_DOUBLE,&count);
      MPI_Type_size(MPI_DOUBLE,&n);
      rdata= (double*) calloc(count,n);
      MPI_Recv(rdata,count,MPI_DOUBLE,0,MPI_ANY_TAG,
                        MPI_COMM_WORLD, &status);
      for(i=0;i<count;i+=10) {
        printf("rdata element %d is %f\n",i,rdata[i]);}
    }
end_time = MPI_Wtime();    /* ends timer */
```

MPI_Probe, MPI_Wtime (C cont'd)


```
 if (myid == 1) {
       printf("Total time is %f x %d\n", end_time-start_time, p);
     }
     MPI_Finalize();                    /* let MPI finish up ... */
}
```

Cartesian Topology
As applied to a 2D Laplace Equation

Laplace Equation

Laplace Equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \qquad (1)$$

Boundary Conditions:

$$\begin{aligned} u(x,0) &= \sin(\pi x) && 0 \le x \le 1 \\ u(x,1) &= \sin(\pi x)e^{-x} && 0 \le x \le 1 \\ u(0,y) &= u(1,y) = 0 && 0 \le y \le 1 \end{aligned} \qquad (2)$$

Analytical solution:

$$u(x,y) = \sin(\pi x)e^{-xy} \qquad 0 \le x \le 1;\ \ 0 \le y \le 1 \qquad (3)$$

Laplace Equation Discretized

Discretize Equation (1) by centered-difference yields:

$$u_{i,j}^{n+1} \cong \frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}}{4} \qquad i = 1,2,\ldots,m; \;\; j = 1,2,\ldots,m \qquad (4)$$

where *n* and *n+1* denote the current and the next time step, respectively, while

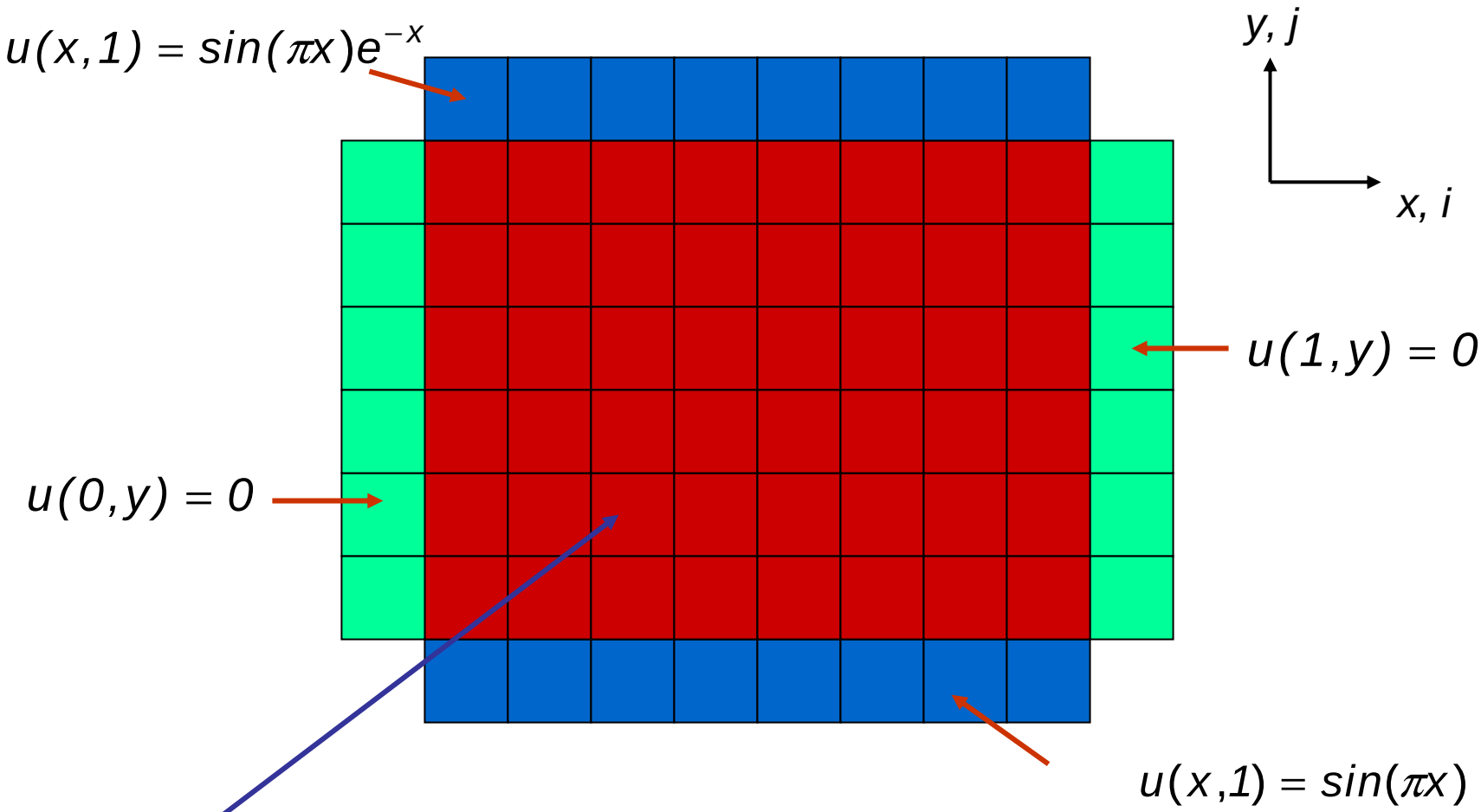
$$u_{i,j}^{n} = u^{n}(x_i, y_j) \qquad i = 0,1,2,\ldots,m+1; \;\; j = 0,1,2,\ldots,m+1 \qquad (5)$$

$$= u^{n}(i\Delta x, j\Delta y)$$

For simplicity, we take

$$\Delta x = \Delta y = \frac{1}{m+1}$$

Computational Domain



$$u_{i,j}^{n+1} \cong \frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}}{4} \qquad i = 1,2,\ldots,m; \quad j = 1,2,\ldots,m$$

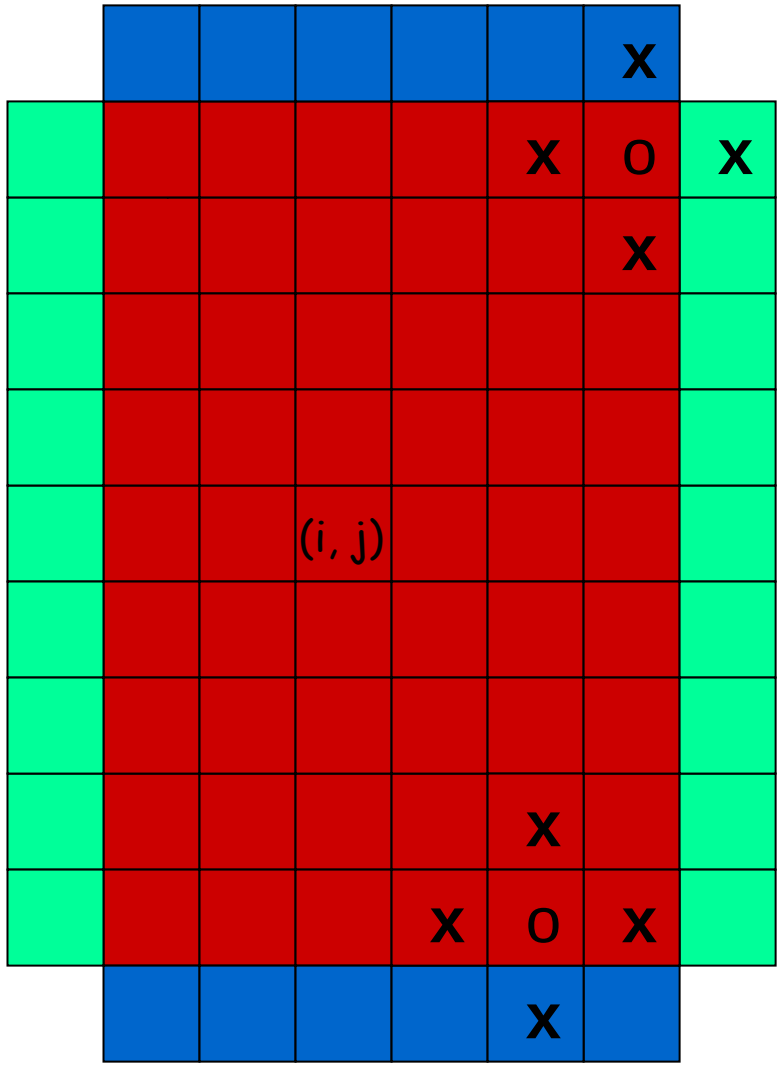
Five-point Finite-Difference Stencil

Interior (or solution) cells.

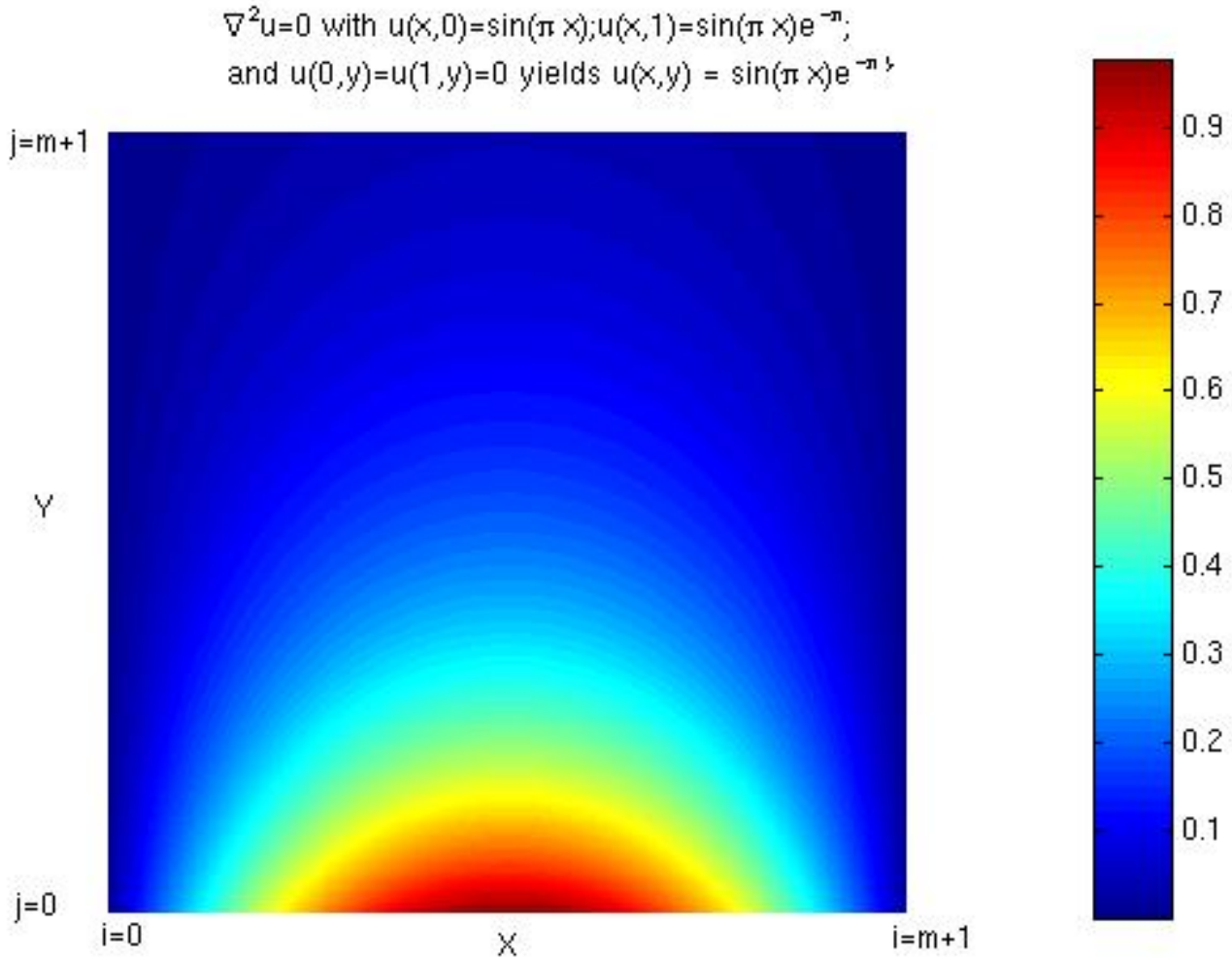
Where solution of the Laplace equation are sought.

Exterior (or boundary) cells.

Green cells denote cells where homogeneous boundary conditions are imposed while non-homogeneous boundary conditions are colored in blue.

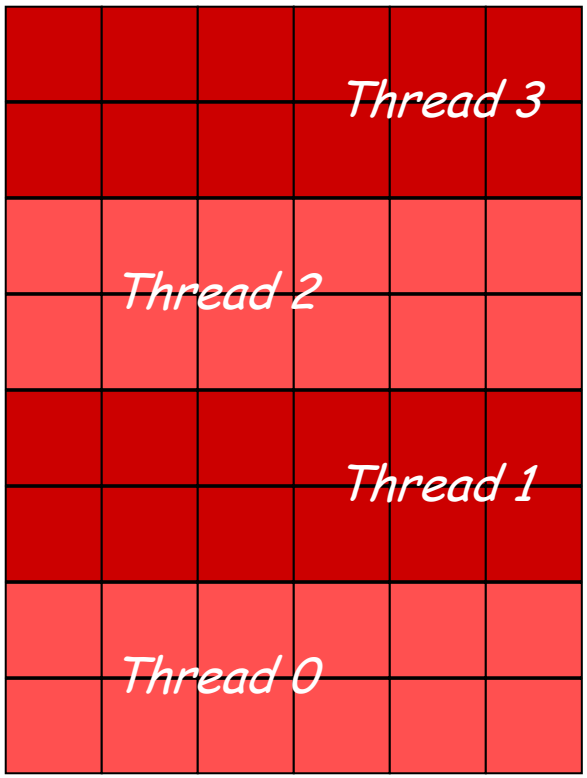


Solution Contour Plot

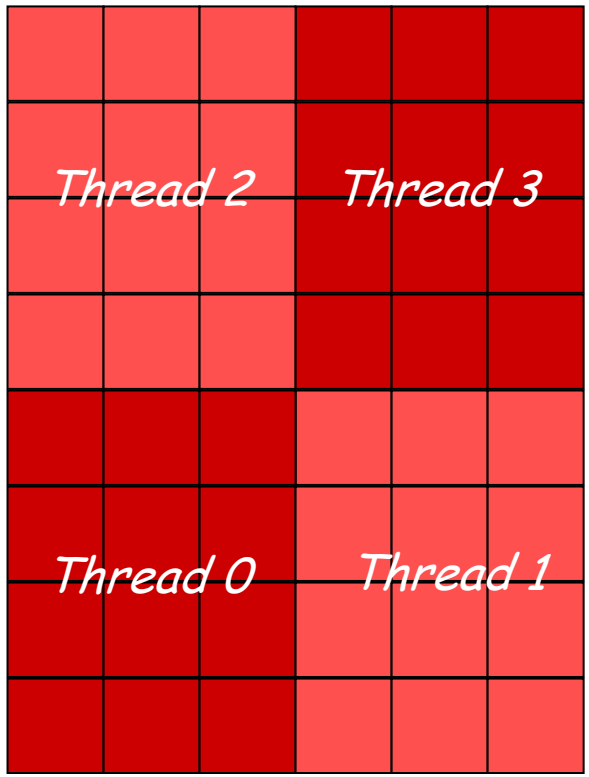


Domain Decompositions

1D Domain Decomposition



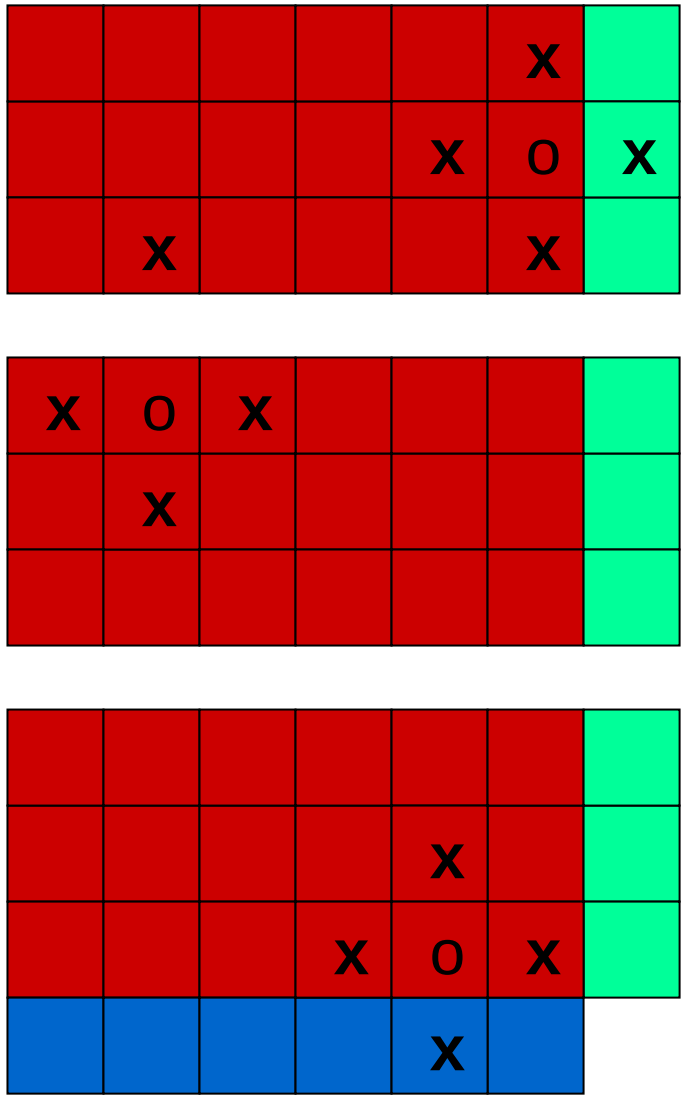
2D Domain Decomposition



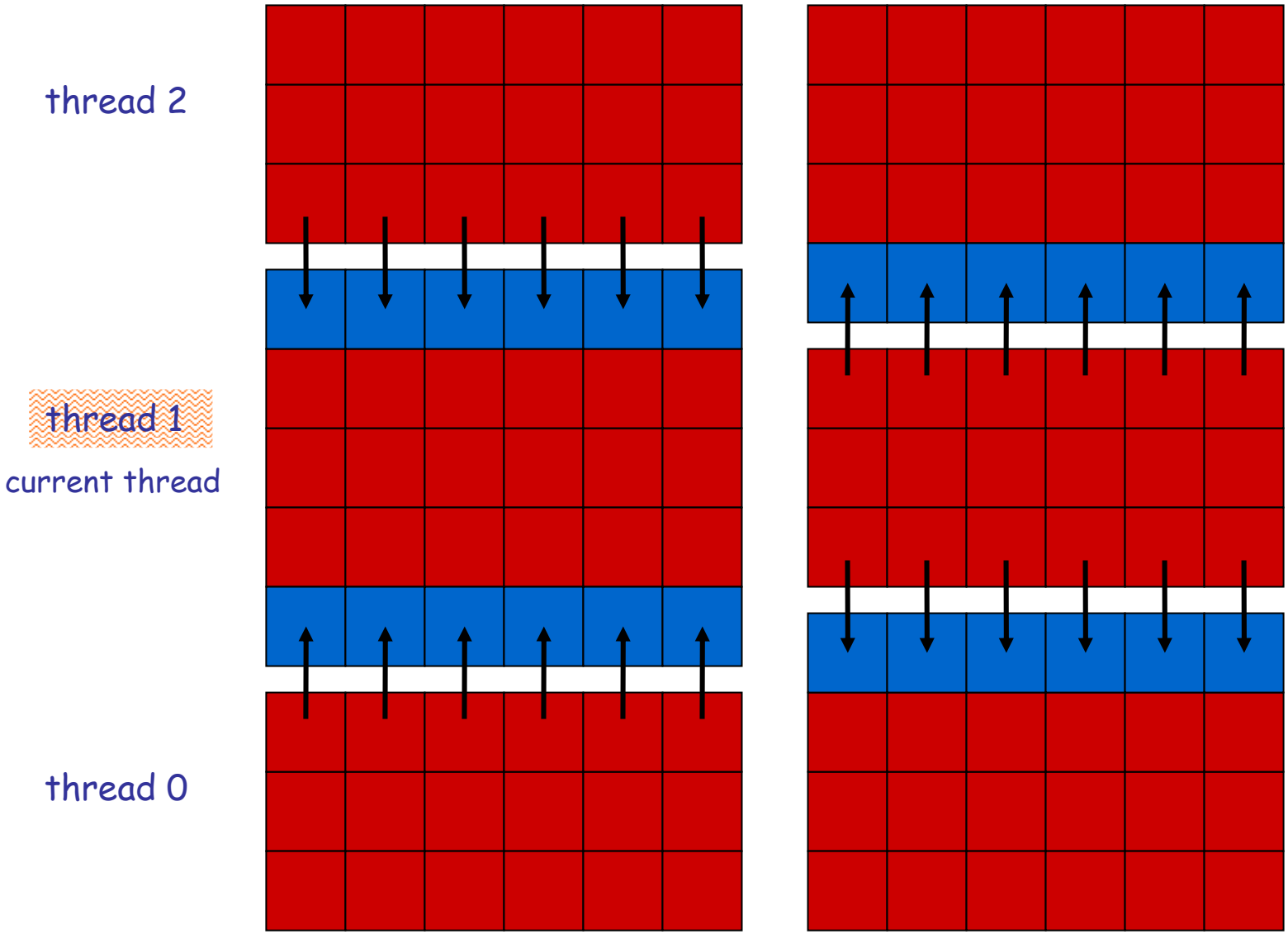
Unknowns At Border Cells – 1D

Five-point finite-difference stencil applied at thread domain border cells require cells from neighboring threads and/or boundary cells.

Message passing required



Message Passing to Fill Boundary Cells



For Individual Threads . . .

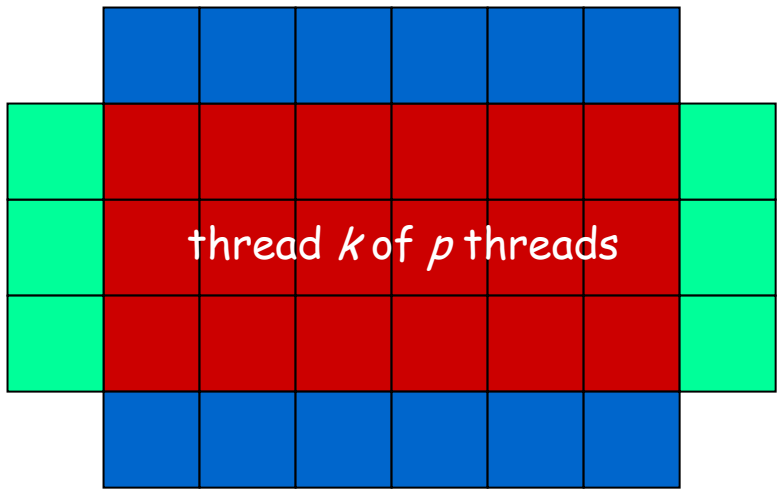
Recast 5-pt finite-difference stencil for individual threads

$$v_{\xi,\eta}^{n+1,k} = \frac{v_{\xi+1,\eta}^{n,k} + v_{\xi-1,\eta}^{n,k} + v_{\xi,\eta+1}^{n,k} + v_{\xi,\eta-1}^{n,k}}{4}$$

$$\xi = 1,2,\ldots,m; \quad \eta = 1,2,\ldots,m'$$
$$m' = m/p; \quad k = 0,1,2,\ldots,p-1$$

Boundary Conditions

- $v_{\xi,m'+1}^{n,k} = v_{\xi,1}^{n,k+1}; \quad \xi = 0,\ldots,m+1; \quad k = 0$
- $v_{\xi,0}^{n,k} = v_{\xi,m'}^{n,k-1}; \quad \xi = 0,\ldots,m+1; \; 0 < k < p-1$
- $v_{\xi,m'+1}^{n,k} = v_{\xi,1}^{n,k+1}; \quad \xi = 0,\ldots,m+1; \; 0 < k < p-1$
- $v_{\xi,0}^{n,k} = v_{\xi,m'}^{n,k-1}; \quad \xi = 0,\ldots,m+1; \; k = p-1$



- For simplicity, assume *m* divisible by *p*
- B.C. time-dependent
- B.C. obtained by message-passing
- Additional boundary conditions on next page

Relationship Between u and v

Physical boundary conditions

- $v_{\xi,0}^{n,k} = u(x_i,0) = sin(\pi x_i); \quad \xi = i = 0,\ldots,m+1; \quad k = 0$
- $v_{\xi,m'+1}^{n,k} = u(x_i,1) = sin(\pi x_i)e^{-\pi}; \quad \xi = i = 0,\ldots,m+1; \; k = p-1$
- $v_{0,\eta}^{n,k} = u(0, y_{\eta+k*m'}) = 0; \quad \eta = 1,\ldots,m'; \; 0 \le k \le p-1$
- $v_{m+1,\eta}^{n,k} = u(1, y_{\eta+k*m'}) = 0; \quad \eta = 1,\ldots,m'; \; 0 \le k \le p-1$

Relationship between global solution *u* and thread-local solution *v*

$$u_{\xi,\eta+k*m'}^{n} = v_{\xi,\eta}^{n,k}$$

$\xi = 1,2,\ \ldots, m; \quad \eta = 1,2,\ \ldots, m'$

$m' = m/p; \quad k = 0,1,2,\ \ldots, p-1$

MPI Functions Needed For Job

- *MPI_Sendrecv* (= *MPI_Send* + *MPI_Recv*) – to set boundary conditions for individual threads
- *MPI_Allreduce* – to search for global error to determine whether convergence has been reached.
- *MPI_Cart_Create* – to create Cartesian topology
- *MPI_Cart_Coords* – to find equivalent Cartesian coordinates of given rank
- *MPI_Cart_Rank* – to find equivalent rank of Cartesian coordinates
- *MPI_Cart_shift* – to find current thread's adjoining neighbor threads

Successive Over Relaxation

1. Make initial guess for *u* at all interior points *(i,j)*.
2. Define a scalar $\omega_n \quad (0 \le \omega_n < 2)$
3. Use 5-pt stencil to compute $u'_{i,j}$ at all interior points *(i,j)*.
4. Compute $u_{i,j}^{n+1} = \omega_n u'_{i,j} + (1-\omega_n) u_{i,j}^{n}$
5. Stop if prescribed convergence threshold is reached.
6. Update: $u_{i,j}^{n} = u_{i,j}^{n+1} \quad \forall \; i, j$
7. Go to step 2.

$$\omega_0 = 0 \;\; ; \;\; \omega_1 = \frac{1}{1-\rho^2/2} \;\; ; \;\; \omega_2 = \frac{1}{1-\rho^2\omega_1/4}$$

$$\omega_n = \frac{1}{1-\rho^2\omega_{n-1}/4}; \quad n > 2$$

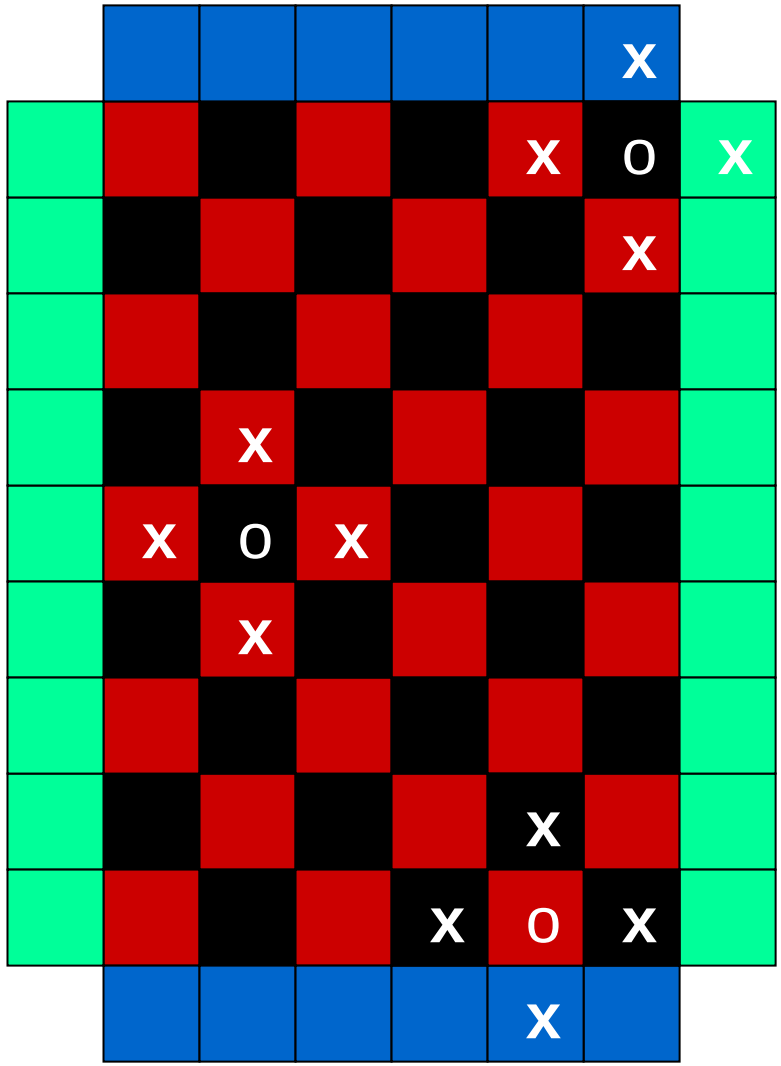
$$\rho = 1 - \left(\frac{\pi}{2(m+1)}\right)^2$$

In Step 3, compute u' with u at time n+1 wherever possible to accelerate convergence. This inhibits parallelism.

Red-Black SOR Scheme

To enable parallelism, note that solution at black cells (by virtue of 5-pt stencil) depend on 4 neighbor red cells. Conversely, red solution cells depend only on 4 respective adjoining black cells.

1. Compute *v* at black cells at time *n+1* in parallel with *v* at red cells at time *n*.
2. Compute *v* at red cells at time *n+1* in parallel with *v* at black cells at time *n+1*.
3. Repeat steps 1 and 2 until converged



Can alternate order of steps 1 and 2.

Which Decomposition: 1D or 2D ?

- Depends on ratio r = problem size / number of processors.
  - For small r, 1D better.
    - Typical thread requires 2 send+receive.
    - Communication overhead dominates.
  - For large r, 2D better (maybe)
    - Typical thread requires 4 send+receive.
    - Smaller perimeter to area ratio means more *local* work per unit message-passing.
- Could implement program that switches from one to the other decomposition pattern as function of r.

MPI Reduction Operations

MPI Reduction Operations:

MPI_Reduce, MPI_Allreduce, MPI_Scan

User must specify a reduction operator when the above reduction operation functions are called. There are two types of reduction operators:

- MPI-defined operators

  MPI_SUM, MPI_MAX, MPI_MIN, …

- User-defined operators

  Subject of this section. But first, a quick review of reduction operations …

Predefined Reduction Operations

Example: Compute $s = \sum_{i=0}^{p-1} i$

Fortran:

```
call MPI_Comm_rank(MPI_COMM_WORLD, i, ierr)
call MPI_Reduce(i, s, 1, MPI_INTEGER, MPI_SUM, dest,  &
                MPI_COMM_WORLD, ierr)
```

C:

```
MPI_Comm_rank(MPI_COMM_WORLD, &i);
MPI_Reduce(i, &s, 1, MPI_INT, MPI_SUM, dest,
                MPI_COMM_WORLD);
```

User-defined Reduction Operations

There are 3 steps to the creation of a user-defined reduction operation

. . .

User-defined Reduction … Step 1

Step 1. The operation, say ⊕, must satisfy the following rules:

- ⊕ must satisfy associative rule $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
  - Addition, “+”, satisfies associative rule.
  - Subtraction, “-”, does not.

- ⊕ *optionally* satisfies the commutative rule $a \oplus b = b \oplus a$
  - Multiplication, “*”, satisfies both associative and commutative rules.
  - Division, “/”, satisfies neither.

User-defined Reduction … Step 2

Step 2. Implement operator into a reduction function following rules:

Fortran:

FUNCTION MYFUNC(IN, INOUT, LEN, DATATYPE)

INTEGER LEN, DATATYPE

<type> IN(LEN), INOUT(LEN)

<type> is one of REAL, INTEGER, COMPLEX, …

DATATYPE is defined by the MPI data type declared in the MPI reduction function call. It should be consistent with <type>.

C:

function myfunc(void *in, void *inout, int *len, MPI_Datatype *datatype)

User-defined Reduction … (cont'd)

Step 3. Register MYFUNC with MPI (*Fortran*) and declare EXTERNAL MYFUNC

```
      EXTERNAL MYFUNC      ! Declare MYFUNC an external function
       INTEGER MYOP        ! MPI handle for MYFUNC
       LOGICAL COMMUTE
          . . .
       COMMUTE = .TRUE.    ! If operator is commutative; else .FALSE.
! Registers MYFUNC with MPI to obtain operator handle MYOP
       CALL MPI_OP_CREATE(MYFUNC, COMMUTE, MYOP, IERR)
       CALL MPI_REDUCE( …, MYOP, …)   ! Use MYOP, not MYFUNC
```

User-defined Reduction … (cont'd)

Step 3. Register myfunc with MPI (*C*)

```
/* Unlike fortran coding, no external declaration for myfunc
     necessary */
     int commute;
     MPI_Op myop;
     commute = 1;          /* if operator is commutative, else 0 */
/* registers myfunc with MPI to obtain operator handle myop */
     MPI_Op_create(myfunc, commute, &myop);
     MPI_Reduce( …, myop, …) ;   /* use myop, not myfunc */
```

Example 2.

A user-implementation of MPI_SUM (*Fortran*)

```
FUNCTION MYSUM(IN, INOUT, LEN, DATATYPE)
INTEGER LEN, DATATYPE, IERR
REAL  IN(LEN), INOUT(LEN)
 INCLUDE 'MPIF.H'
 IF (DATATYPE .NE. MPI_REAL)
&        CALL MPI_ABORT(MPI_COMM_WORLD, 1, IERR)
 DO I=1,LEN
   INOUT(I) = INOUT(I) + IN(I)
 ENDDO
 END
```

Example 3. One-norm

Various norms are often used to measure the convergence history of numerical solutions. One-norm is defined as

$$N_1(\vec{x}) = \sum_{j=0}^{p-1} | x_j |$$

- MPI_SUM could be used with MPI_Reduce to achieve the effect of one-norm
- “+” is the reduction operator
- Will implement one-norm to highlight the computing procedure of processes

Example 3. One-norm subroutine

A user-implementation of one-norm (*Fortran*)

```
FUNCTION ONENORM(IN, INOUT, LEN, DATATYPE)
 INTEGER LEN, DATATYPE
 REAL  IN(LEN), INOUT(LEN)

 DO I=1,LEN
   INOUT(I) = ABS(INOUT(I)) + ABS(IN(I))
 ENDDO
 END
```

8-procs One Norm Example